NASA Contractor Report 189588

Mechanically Verified Hardware Implementing an 8-Bit Parallel IO Byzantine Agreement Processor

J Strother Moore

Computational Logic, Inc.
Austin, Texas

Contract NAS1-18878
1992

Abstract

Consider a network of four processors that use the Oral Messages (Byzantine Generals) algorithm of Pease, Shostak and Lamport to achieve agreement in the presence of faults. Bevier and Young have published a functional description of a single processor that, when interconnected appropriately with three identical others, implements this network under the assumption that the four processors step in synchrony. By formalizing the original Pease, Shostak and Lamport work, Bevier and Young mechanically proved that such a network achieves fault tolerance. In this paper we develop, formalize and discuss a hardware design that has been mechanically proved to implement their processor. In particular, we formally define mapping functions from the abstract state space of the Bevier-Young processor to a concrete state space of a hardware module and state a theorem that expresses the claim that the hardware correctly implements the processor. We briefly discuss the Brock-Hunt Formal Hardware Description Language which permits designs both to be proved correct with the Boyer-Moore theorem prover and to be expressed in a commercially supported hardware description language for additional electrical analysis and layout. We briefly describe our implementation, which actually takes the form of a hardware design generator which produces a design as a function of the desired word size. We exhibit the theorem that establishes that the generator is correct. We exhibit the instance generated for sense data of width 8, in the syntax of NDL, a hardware description language supported by LSI Logic, Inc. We exhibit some results of processing the verified design with commercially available tools. We discuss two unrealistic aspects of our verified design. (a) The use of parallel instead of serial io requires an excessive number of pins. (b) The assumption that all four processors step in synchrony is implemented by having them share a common clock— introducing an unacceptable single-point failure mode.

Keywords: hardware verification, fault tolerance, Byzantine agreement, Oral Messages algorithm, automatic theorem proving, Boyer-Moore logic.

1. Background

In [1] Bevier and Young describe a formalization of the ‘‘Oral Messages’’ (or ‘‘Byzantine Generals’’) algorithm of Pease, Shostak and Lamport [5] and a functional description of a processor that implements the algorithm in the case of a four processor network. They use the Boyer-Moore theorem prover, NQTHM [2], to check the Pease-Shostak-Lamport theorem and to prove that their abstract processor correctly implements the algorithm for the case in question. They specify the processor by exhibiting a function named **`local-step`** that is the state transition function, i.e., the function that, on each clock tick, produces the next state of the processor. In this paper we implement that processor in the formalized hardware description language (HDL) of Brock and Hunt and we exhibit a theorem, which has been proved by NQTHM, that states that our hardware implements **`local-step`**. Readers are urged to see [1] for additional background material.

The processor reads sense data and inputs from its peers, exchanges this data in a certain fixed pattern among the peers, and then votes on certain combinations of the exchanged data. The result of the vote is an ‘‘interactive consistency vector’’ (‘‘icv’’) which contains four data objects in 1:1 correspondence with the four processors. The icv in a processor indicates that processor’s ‘‘opinion’’ of the final value of the sense data in each of the four. Provided at most one processor is faulty, all nonfaulty processors hold identical opinions about all the processors, including any faulty processor. This fact is proved informally but precisely in [5]; it is stated formally and proved mechanically in [1]. In actual applications, the sense data and all of the exchanged data are in fact bit vectors of some fixed length, though that restriction is unnecessary in the abstract view of the processor and in its proof.

Bevier and Young formalize the processor by formalizing the notion of its ‘‘state’’ and its ‘‘state transition’’ function, the function which determines the next state given the sense data, the input from the peers, and the current state. To model the network in which the four processors are connected, Bevier and Young define a function called **`global-step`** which manages four independent processor states and transfers the outputs of each state to the appropriate inputs of the next state transition. This model of the network implicitly assumes that all four processors execute in lockstep synchrony. If **`local-step`** is taken as a low-level hardware design, in which one state transition by **`local-step`** describes one tick of the microprocessor’s clock, then this assumption is naturally implemented by having the four processors controlled by a common clock. If **`local-step`** is taken more abstractly, in which one step by **`local-step`** might require many microprocessor cycles, then this assumption might be implemented via some rough clock synchronization algorithm and time abstraction. We take the view that **`local-step`** is a low-level specification and we designed our microprocessor to implement it directly. This is unrealistic for two reasons. First, it requires the four processors to share a common clock, which introduces a potential single-point failure mode. Second, it requires parallel io so that all the bits output by one processor on one clock tick are available as input to the appropriate peer processor on its next cycle. But because we have so many inputs and outputs, parallel io makes excessive demands for pins. We return to these points after presenting our design.

The state of the abstract processor, **`local-step`**, is a 5-tuple constructed by **`state`** from

- a 3×3 **`matrix`** of sense data read and obtained from peers;
- an output buffer **`obuf`** of length 3, each component of which is physically connected to a fixed peer processor in such a way that the contents of that component on each cycle appears as a certain input to the peer on its next cycle;
- the interactive consistency vector **`icv`** containing data objects (or a token denoting "no majority") representing the finally agreed upon values of the sense data in each of the four peers;
- a **`light`** which represents the final action taken by the processors upon reaching agreement;

and

- a counter, **`clock`**, which records the current ‘‘time’’ modulo 8 and is used to sequence the device.

Bevier and Young define the notion of a ‘‘good state’’ with **`good-statep`** which formalizes the description above. See Appendix A.

The definition of **`local-step`** is

Definition.
```
(local-step input state)
   =
(let ((sense (nth 0 input))
      (p0 (nth 1 input))
      (p1 (nth 2 input))
      (p2 (nth 3 input))
      (clock (clock state)))
  (case (remainder clock 8)
        (0 (state (matrix state)
                  (make-list 3 sense)
                  (put 3 sense (icv state))
                  (light state)
                  (remainder (plus 1 (clock state)) 8)))
        (1 (state (put 0 (list p0 p1 p2) (matrix state))
                  (list p1 p0 p0)
                  (icv state)
                  (light state)
                  (remainder (plus 1 (clock state)) 8)))
        (2 (state (put 1 (list p0 p1 p2) (matrix state))
                  (list (nth 2 (nth 0 (matrix state)))
                        (nth 2 (nth 0 (matrix state)))
                        (nth 1 (nth 0 (matrix state))))
                  (icv state)
                  (light state)
                  (remainder (plus 1 (clock state)) 8)))
        (3 (state (put 2 (list p0 p1 p2) (matrix state))
                  (obuf state)
                  (icv state)
                  (light state)
                  (remainder (plus 1 (clock state)) 8)))
        (4 (state (matrix state)
                  (obuf state)
                  (compute-icv (matrix state) (icv state))
                  (light state)
                  (remainder (plus 1 (clock state)) 8)))
        (5 (state (matrix state)
                  (obuf state)
                  (icv state)
                  (filter (icv state))
                  (remainder (plus 1 (clock state)) 8)))
        (otherwise
         (state (matrix state)
                (obuf state)
                (icv state)
                (light state)
                (remainder (plus 1 (clock state)) 8))))).
```

The **`case`** and **`let`** abbreviations (supported by some local patches to NQTHM) should be self-explanatory. The definition of **`local-step`** without these abbreviations may be found in Appendix A, along with the definitions of the subfunctions. Roughly speaking, the function above produces a new state as a function of the current state and the input. On each application, the clock is incremented by one (modulo 8). When the clock is between 0 and 5, other components of the state are modified. The last two cycles (6 and 7) are no-ops.

Our job is to construct a Formal HDL description of a module that implements this function and to prove that we did so.

The Formal HDL we use is the descendant of that described by Brock and Hunt in [3]. (At the time of this writing, the new Formal HDL has not yet been documented though we explain it briefly here.) The language is connected to the hardware design tools of LSI Logic, Inc., via a Lisp program that translates Formal HDL descriptions into LSI Logic's Netlist Description Language (NDL). NDL is a conventional hardware description language similar to Verilog™ [7]. Commercially available LSI Logic tools permit one to analyze NDL descriptions to extract schematics, do layout, etc.

In this document we exhibit our implementation and the theorem that we claim establishes its correctness. We sketch the Formal HDL to make our description somewhat self-contained, but we do not include the definition of the HDL, nor do we even discuss (much less present the NQTHM events leading to) the proof of correctness. However, the file of events, leading from NQTHM's **`ground-zero`** state through the definition of the Brock-Hunt hardware interpreter, **`dual-eval`**, and thence onward to our implementation of **`local-step`** and its correctness, is available upon request. The file may be processed by the released NQTHM, but requires the loading of Bishop Brock's ‘‘fast clausifier’’ patch, available from CLI.

2. Mapping from Abstract States to Concrete States

The function we wish to implement, **`local-step`**, uses such abstract objects as the 5-tuple **`state`**s, integers, the arbitrary sense data objects, etc. In order to implement it in digital hardware we must both restrict it to certain kinds of sense data (e.g., bit vectors) and define a mapping from the abstract state space to a concrete state space.

The hardware description language we use imposes on us a formal definition of concrete states as **`cons`**-trees of Boolean vectors. The shape of the tree depends on the hierarchical decomposition of the hardware description. Thus, our description of the concrete state space foreshadows our final implementation. Nevertheless, we describe the two state spaces (the restricted abstract one and the concrete one) and the maps between them before exhibiting our implementation.

2.1 The Restricted Abstract State Space

Brock and Hunt define a bit vector to be recognized by **`bvp`**,

Definition.
```
(bvp x)
   =
(if (nlistp x)
    (equal x 'nil)
    (and (boolp (car x)) (bvp (cdr x)))).
```

We introduce the idea of a bit vector of width **`w`**,

Definition.
```
(bvpn bv w)
   =
(and (bvp bv)
     (equal (length bv) w)),
```

and the idea of a proper list of such bit vectors,

Definition.

```
(all-bvpn lst w)
    =
(if (nlistp lst)
    (equal lst nil)
    (and (bvpn (car lst) w)
         (all-bvpn (cdr lst) w)))
```

In our restricted abstract states, sense data (and thus the exchanged and voted data) will always be bit vectors of width `w`.

The icv of the abstract state will be restricted to being a list of length 4, the last element of which is a bit vector of width `w` and the other three of which are either bit vectors of width `w` or else the object `(maj-token)` indicating that no majority was found.

Definition.

```
(icvp lst w)
    =
(and (equal (length lst) 4)
     (or (bvpn (car lst) w)
         (equal (car lst) (maj-token)))
     (or (bvpn (cadr lst) w)
         (equal (cadr lst) (maj-token)))
     (or (bvpn (caddr lst) w)
         (equal (caddr lst) (maj-token)))
     (bvpn (cadddr lst) w)
     (equal (cddddr lst) nil))
```

Similarly, we require that the matrix and the output buffer of the restricted abstract state contain fixed width bit vectors. We wrap all these restrictions up into a single predicate,

Definition.

```
(data-path-assumptionp state w)
    =
(and (properp (matrix state))
     (all-bvpn (nth 0 (matrix state)) w)
     (all-bvpn (nth 1 (matrix state)) w)
     (all-bvpn (nth 2 (matrix state)) w)
     (all-bvpn (obuf state) w)
     (icvp (icv state) w)).
```

The abstract state contains the `light` which is set by the undefined function `filter` once the icv is computed. We cannot implement either the `light` or the `filter` in hardware since they are unspecified. Therefore, the map up from concrete states to abstract ones must somehow recover from the concrete state an icv upon which `filter` produces the required `light`. Thus, if we are originally presented with an abstract state whose `light` is not the value of `filter` on some icv it will be impossible to map the state down invertibly. We therefore impose on the abstract state space the additional restriction that the `light` of an abstract state be obtained by applying `filter` to some `icvp`-object. We avoid the implicit existential quantification by passing the alleged object in as a ‘‘witness.’’ States for which there exists such a witness are said to be ‘‘well-lit.’’

Definition.

```
(well-litp state act-reg w)
    =
(and (icvp act-reg w)
     (equal (light state) (filter act-reg)))
```

Our restricted abstract state space is defined by

Definition.
```
(bevier-young-statep state act-reg w)
   =
(and (good-statep state)
     (data-path-assumptionp state w)
     (well-litp state act-reg w)),
```

which recognizes well-lit good states that satisfy the data path width assumption.

2.2 The Concrete State Space

The shape of a concrete state is determined by the hierarchical decomposition of our hardware description and the conventions of the Formal HDL language. A concrete state will be a list of length nine with the following components, arrayed in the order shown:

component	structure	types
`cnt`	`(c0 c1 c2)`	three bits
`matrix0`	`(m00 m01 m02)`	three w-bit vectors
`matrix1`	`(m10 m11 m12)`	three w-bit vectors
`matrix2`	`(m20 m21 m22)`	three w-bit vectors
`data-out`	`(o0 o1 o2)`	three w-bit vectors
`icv-reg`	`(icv0 icv1 icv2 icv3)`	four w-bit vectors
`act-reg`	`(a0 a1 a2 a3)`	four w-bit vectors
`icv-maj-existsp-reg`	`(b0 b1 b2)`	three bits
`act-maj-existsp-reg`	`(d0 d1 d2)`	three bits

For example, `(cnt state)` is defined to be `(nth 0 state)` and `(act-maj-existsp-reg state)` is defined to be `(nth 8 state)`.

The recognizer for well-formed concrete states is:

Definition.
```
(hunt-brock-statep x w)
   =
(and (properp x)
     (equal (length x) 9)
     (bvpn (cnt x) 3)
     (equal (length (matrix0 x)) 3)
     (equal (length (matrix1 x)) 3)
     (equal (length (matrix2 x)) 3)
     (equal (length (data-out x)) 3)
     (equal (length (icv-reg x)) 4)
     (equal (length (act-reg x)) 4)
     (all-bvpn (matrix0 x) w)
     (all-bvpn (matrix1 x) w)
     (all-bvpn (matrix2 x) w)
     (all-bvpn (data-out x) w)
     (all-bvpn (icv-reg x) w)
     (all-bvpn (act-reg x) w)
     (bvpn (icv-maj-existsp-reg x) 3)
     (bvpn (act-maj-existsp-reg x) 3))
```

2.3 The Map Down

We map the abstract icv onto the concrete icv by replacing **`(maj-token)`**, when it occurs, by the bit vector that is everywhere **`f`**. This is ambiguous, since that bit vector may be legitimate sense data. We therefore maintain a 3-bit register, **`maj-existsp-reg`**, which is in 1:1 correspondence with the first three words of the concrete icv and in which an **`f`** indicates that the corresponding icv word denotes **`(maj-token)`**.

Definition.
```
(icv-down icv w)
    =
(list (if (equal (car icv) (maj-token))
          (nat-to-v 0 w)
          (car icv))
      (if (equal (cadr icv) (maj-token))
          (nat-to-v 0 w)
          (cadr icv))
      (if (equal (caddr icv) (maj-token))
          (nat-to-v 0 w)
          (caddr icv))
      (cadddr icv))
```

Here is how we set **`maj-existsp-reg`** from the abstract icv:

Definition.
```
(maj-existsp-reg icv)
    =
(list (not (equal (car icv) (maj-token)))
      (not (equal (cadr icv) (maj-token)))
      (not (equal (caddr icv) (maj-token))))
```

To map an abstract state down (invertibly) we must know the witness for the **`light`** of the abstract state. This witness, which is another icv, we store in the **`act-reg`** and the **`act-maj-existsp-reg`** (affording the **`(maj-token)`**s in the witness the same treatment as in the icv). Thus, we map an abstract state down (with respect to a given witness **`act-reg`** and data width **`w`**) with

Definition.
```
(down state act-reg w)
    =
(list (nat-to-v (clock state) 3)
      (nth 0 (matrix state))
      (nth 1 (matrix state))
      (nth 2 (matrix state))
      (obuf state)
      (icv-down (icv state) w)
      (icv-down act-reg w)
      (maj-existsp-reg (icv state))
      (maj-existsp-reg act-reg)).
```

2.4 The Map Up

We invert the **`icv-down`** map with

Definition.
```
(icv-up icv icv-maj-existsp-reg)
    =
(list (if (car icv-maj-existsp-reg) (car icv) (maj-token))
      (if (cadr icv-maj-existsp-reg) (cadr icv) (maj-token))
      (if (caddr icv-maj-existsp-reg) (caddr icv) (maj-token))
      (cadddr icv)),
```

which is also used to recover the **`light`** witness.

We then invert the **`down`** map with

Definition.

```
(up lst)
   =
(state (list (matrix0 lst) (matrix1 lst) (matrix2 lst))
       (data-out lst)
       (icv-up (icv-reg lst)
               (icv-maj-existsp-reg lst))
       (filter (icv-up (act-reg lst)
                       (act-maj-existsp-reg lst)))
       (v-to-nat (cnt lst)))
```

Observe that after recovering the witness from **`act-reg`** and **`act-maj-existsp-reg`** we apply **`filter`** to obtain the **`light`**.

3. The Specification

In the most literal sense, our goal is to exhibit a netlist that implements the Bevier-Young **`LOCAL-STEP`** for some fixed data path width, **`w`**, namely 8. We exhibit such a netlist in Appendix C, where it is displayed in the syntax of LSI Logic's NDL. Let *netlist* be the formal analogue of that netlist, let *module* be the formal analogue of the top-level module name, **`LSTEP_8`**, and let *w* be the data width 8. Then the following theorem holds:

Theorem. **`Main`**

```
(implies (and (bevier-young-statep state act-reg w)
              (bvpn sense w)
              (bvpn p0 w)
              (bvpn p1 w)
              (bvpn p2 w)
              (equal te f)
              (equal reset- t))
         (equal (local-step (list sense p0 p1 p2) state)
                (up
                 (dual-eval 2 module
                            (append (list clk te ti reset-)
                                    (append sense
                                            (append p0
                                                    (append p1 p2))))
                            (down state act-reg w)
                            netlist))))
```

This theorem says that if

- **`state`** is a state in our restricted abstract state space (whose **`light`** is witnessed by **`act-reg`** and whose data path width is *w*), and
- **`sense`**, **`p0`**, **`p1`**, **`p2`** are bit vectors of width *w*, and
- the ''test enable'' line, **`te`**, to our module is low and the ''reset when low'' line, **`reset-`**, is high,

then

- the Bevier-Young **`local-step`** applied to the given sense and input data in the given abstract state

is equal to the result of

- mapping the abstract state down to a concrete state (using the supplied witness),
- stepping the Brock-Hunt hardware model forward one step on that concrete state with our given inputs, *module*, and *netlist*, and
- mapping the resulting concrete state back up.

Technically speaking, we do not actually cause NQTHM to prove this theorem. We actually define both a netlist generator and a netlist recognizer, both of which take the data width, **`w`**>0, as a parameter. The generator produces a list constant that is the formal HDL description of a netlist that implements **`local-step`** for the given data width. The recognizer returns **`t`** or **`f`** according to whether a given netlist is some extension of the one we generate for **`w`**. We then lead NQTHM to the proof of the theorem that if the recognizer accepts a **`netlist`** defining **`module`** for width **`w`**>0 (where **`netlist`**, **`module`** and **`w`** are now universally quantified variables) then the interpretation of the module under the netlist computes **`local-step`** in the sense illustrated above. We do not *prove* that our generator always constructs a netlist satisfying the recognizer. Rather, we merely execute the generator on any chosen *w*, obtain a concrete *netlist*, and then execute the recognizer on that *netlist* to observe that the generator worked for that particular *w*. This is faster than proving that the generator always satisfies the recognizer, since one usually only generates a small number of instances of the design.

4. The Implementation

Our implementation is decomposed into modules. We exhibit the module definition generators in Appendix B. In this section we explain a few of the modules simply to illustrate the HDL and our implementation.

4.1 Incrmt3

The following NQTHM function defines the implementation of the **`INCRMT3`** module. The module takes three bits in, **`i0`**...**`i2`**, and produces three bits, **`o0`**...**`o2`**. If the two bit vectors are thought of as integers in binary notation, then the specification of this module is that the output is the successor of the input, modulo 8. We state this specification formally later. The implementation defines the output, in terms of the input, with combinational logic: **`o0`** is the logical negation of **`i0`**; **`o1`** is the exclusive-or of **`i0`** and **`i1`**; and **`o2`** is the exclusive-or the intermediate signal **`s0`** and **`i2`**, where **`s0`** is the conjunction of **`i0`** and **`i1`**.

Definition.
```
(incrmt3*)
   =
'(incrmt3 (i0 i1 i2) (o0 o1 o2)
          ((g0 (o0) b-not (i0))
           (g1 (o1) b-xor (i0 i1))
           (g2 (s0) b-and (i0 i1))
           (g3 (o2) b-xor (s0 i2)))
          nil)
```

The module definition is a list of five parts. The first part, **`incrmt3`**, is the name of the module. The second part, **`(i0 i1 i2)`**, is the list of input signals. The third part, **`(o0 o1 o2)`**, is the list of output signals. The fourth part is a list of ''occurrences,'' each of which is a list of the form **`(occ-name output mod-name input)`** meaning that the signals listed in the **`output`** list are those produced by the module **mod-name** with input **input** in the current state. The occurrence names, **occ-name**, e.g., **g0**, **g1**, etc., are irrelevant here. The fifth part of a module definition is the list of state-holding occurrences. In the module above there are none so the list is **nil**. Note that the ‘‘submodules’’ of **incrmt3** (the modules used in its definition) are **b-not**, **b-xor**, and **b-and**. These are all primitive but in general they may be the name of other defined modules.

Incrmt3* can be thought of as a parameterized module generator that happens to have no parameters (and thus is a constant). Many of our module generators take arguments that indicate the size of the data, say, and use list processing functions to construct a suitable module definition. All of our module generators have names that end in *****.

In addition to its module definition generator, each module is associated with two other functions, a netlist generator and a netlist recognizer. A netlist is just a list of module definitions. The netlist generator for a module produces a list containing the definition of the module and all of its submodules. The netlist generator for the **incrmt3** module is shown below.

Definition.
```
(incrmt3$netlist)
   =
(cons (incrmt3*)
      (union (b-not$netlist)
             (union (b-and$netlist)
                    (b-xor$netlist))))
```

All of our netlist generators have names that end in **$netlist**.

The netlist recognizer for **incrmt3** recognizes when a given netlist contains the definition of **incrmt3** and all of its submodules.

Definition.
```
(incrmt3& netlist)
   =
(and (equal (lookup-module 'incrmt3 netlist) (incrmt3*))
     (and (b-not& (delete-module 'incrmt3 netlist))
          (and (b-and& (delete-module 'incrmt3 netlist))
               (b-xor& (delete-module 'incrmt3 netlist)))))
```

All of our netlist recognizers have names that end in **&**. Because the netlist generators and recognizers can be deduced from the module definitions, we henceforth discuss only the module definition generators.

To specify and prove the correctness of modules we must have a way of formally deriving their outputs and state changes from their inputs and their definitions. Bishop and Hunt define the NQTHM function **dual-eval** which can be thought of as an interpreter for their HDL. **Dual-eval**’s first argument is a flag that determines whether the function returns the signals output by the module or the new state created by the module. The signal values are returned if the flag is **0** and the state value is returned if the flag is **2**. Other values of the flag have other meanings.

Using **dual-eval** we can state the correctness of **incrmt3**.

Theorem.
```
(implies (and (incrmt3& netlist)
              (bvpn i 3))
         (equal (dual-eval 0 'incrmt3 i state netlist)
                (nat-to-v (add1 (v-to-nat i)) 3)))
```

This theorem says that if **netlist** contains the definition of **incrmt3** and its submodules and **i** is a bit vector of length 3, then the output produced by evaluating the **incrmt3** module with input **i**, in any state, is obtained by converting **i** to a natural number, incrementing it by one, and converting the result into a 3-bit vector. This formula has been proved by NQTHM.

Observe that if we proceeded to use **incrmt3** as a submodule in some other module, and then tried to prove that module correct, the netlist recognizer for that superior module would insure that the netlist recognizer for **incrmt3** were satisfied. Hence, if during the symbolic evaluation of that superior module the question arose ‘‘what is the value of the **incrmt3** module on **x0**...**x2**?’’ the answer is provided by the correctness theorem for **incrmt3** above. Thus, this methodology lets us ‘‘stack’’ modules and their correctness theorems to build complex structures.

We can run a Common Lisp function on the definition of **incrmt3** to translate it into NDL. The result is

```
MODULE INCRMT3;
INPUTS I0,I1,I2;
OUTPUTS O0,O1,O2;
LEVEL FUNCTION;
DEFINE
G0(O0) = IVA(I0);
G1(O1) = EO(I0,I1);
G2(S0) = AN2(I0,I1);
G3(O2) = EO(S0,I2);
END MODULE;
```

Note that the ‘‘deep’’ structure of the definition is identical. The primitive module names (for ‘‘invert,’’ ‘‘exclusive or’’ and ‘‘and’’) are those supported by LSI Logic’s design tools

We can process this NDL description of the module with LSI Logic’s ‘‘schematic liberator’’ and obtain the mechanically drawn schematic diagram included in Appendix D.

4.2 Counter3

The following module, **counter3-temp**, merely conjoins the **reset-** signal with each of its other three inputs. The three output signals of **counter3-temp** are thus **f** if **reset-** is **f** and are otherwise just the three input signals.

Definition.

```
(counter3-temp*)
   =
'(counter3-temp (reset- i0 i1 i2)
                (d0 d1 d2)
                ((g0 (d0) b-and (reset- i0))
                 (g1 (d1) b-and (reset- i1))
                 (g2 (d2) b-and (reset- i2)))
                nil)
```

We use **incrmt3**, **counter3-temp** and our first state-holding device to construct **counter3**.

Definition.

```
(counter3*)
   =
(list 'counter3
      '(clk te ti reset-)
      (indices 'q 0 3)
      (list (list 'reg
                  (indices 'q 0 3)
                  (index 'reg 3)
                  (cons 'clk
                        (cons 'te
                              (cons 'ti (indices 'd 0 3)))))
            (list 'inc
                  (indices 'i 0 3)
                  'incrmt3
                  (indices 'q 0 3))
            (list 'g0
                  (indices 'd 0 3)
                  'counter3-temp
                  (cons 'reset- (indices 'i 0 3))))
      'reg)
```

The expression **`(index name i)`** constructs an indexed name, e.g., **`(index 'reg 3)`** may be thought of as **`'REG3`**. The expression **`(indices name 0 k)`** constructs the list of **`k`** consecutive indexed names starting from index **`0`**.

Thus, the module above takes four input signals, **`clk`**, **`te`**, **`ti`** and **`reset-`**, and produces three output signals, **`q0`**...**`q2`**. The first three input signals are used in LSI Logic's low level register module for the clock, the test enable line, and the test input line. The two test signals allow us to chain registers together so as to load and read the state of a module serially. In our designs we use the **`te`** and **`ti`** inputs to build such ''scan chains.'' But we do not discuss them here and we have not proved that our scan chains work; all our theorems contain the hypothesis that the **`te`** signal is **`f`**, which means that our theorems address themselves only to the behavior of our modules in non-test mode.

The first occurrence above, the one named **`reg`**, says that the three output signals, **`q0`**...**`q2`**, are obtained from the module **`reg3`** by giving it the six inputs **`clk`**, **`te`**, **`ti`**, **`d0`**...**`d2`** in the current state. The **`reg3`** module is a primitive module for a state-holding device of width 3. Its value is just the contents of the current state (modulo the **`te`** and **`ti`** inputs which we do not further discuss). But the new state delivered by **`reg3`** is the list of three signals, **`d0`**...**`d2`**. Thus, this first occurrence sets the module output to the three signals in the current state and makes the new state be **`d0`**...**`d2`**. But we have not defined these signals yet.

The second occurrence above, the one named **`inc`**, should be read ''Let **`i0`**...**`i2`** be obtained by incrementing **`q0`**...**`q2`** with the **`incrmt3`** module. Thus, the **`i0`**...**`i2`** represent the number one greater (modulo 8) than the value returned by **`counter3`**.

Finally, the third occurrence above, named **`g0`**, defines **`d0`**...**`d2`** to be the result of applying **`counter3-temp`** to the **`reset-`** signal and **`i0`**...**`i2`**.

Note that the fifth part of the module definition above is **`'reg`**. This is the single occurrence of a state-holding device in the module and it describes the state returned by this module. In this case, the state is just the 3-bit state of the **`reg3`** module. In general the fifth part of a module definition is either a single occurrence name or a (possibly empty) list of occurrence names.

Functionally, the **`counter3`** module can be thought of as operating on four signal arguments and a 3-bit state and producing three signal values and a 3-bit state. The signals returned are just those in the state in which **`counter3`** is evaluated. The state returned is obtained by incrementing its current state by one (modulo 8) and zeroing it if **`reset-`** is **`f`**. This specification of **`counter3`** is captured in the two theorems shown below.

Theorem.
```
(implies (and (counter3& netlist)
              (equal te f)
              (bvp cnt)
              (equal (length cnt) 3))
         (equal (dual-eval 0 'counter3
                             (list clk te ti reset-)
                             cnt netlist)
                cnt))
```

Theorem.
```
(implies (and (counter3& netlist)
              (equal te f)
              (boolp reset-)
              (bvp cnt)
              (equal (length cnt) 3))
         (equal (dual-eval 2 'counter3
                             (list clk te ti reset-)
                             cnt netlist)
                (if reset-
                    (nat-to-v (add1 (v-to-nat cnt)) 3)
                    (list f f f))))
```

The first specifies the signals returned by **`counter3`** and the second specifies the state returned.

The NDL for the two modules is

```
MODULE COUNTER3-TEMP;
INPUTS RESET-,I0,I1,I2;
OUTPUTS D0,D1,D2;
LEVEL FUNCTION;
DEFINE
G0(D0) = AN2(RESET-,I0);
G1(D1) = AN2(RESET-,I1);
G2(D2) = AN2(RESET-,I2);
END MODULE;

MODULE COUNTER3;
INPUTS CLK,TE,TI,RESET-;
OUTPUTS Q.0,Q.1,Q.2;
LEVEL FUNCTION;
DEFINE
REG(Q.0,Q.1,Q.2) = REG_3(CLK,TE,TI,D.0,D.1,D.2);
INC(I.0,I.1,I.2) = INCRMT3(Q.0,Q.1,Q.2);
G0(D.0,D.1,D.2) = COUNTER3-TEMP(RESET-,I.0,I.1,I.2);
END MODULE;
```

4.3 Other Submodules

Our implementation of **`local-step`** uses six modules in addition to the three explained above. We merely describe them here. The corresponding module definitions are shown in Appendix B.

`Split-3-to-6` takes the three bits returned by the **`counter3`** module, which correspond to **`local-step`**'s clock, and returns six signals, **`s0`**...**`s5`**, with the property that **`s`***i* is **`t`** iff the three input bits represent the number *i* in binary. This module is a demultiplexor. When the clock is 6 or 7, all the output signals are **`f`**.

`Majority3` is a module parameterized by the data width `n`. It takes three `n`-bit vectors in. It delivers a single bit, called `maj-existsp`, and an `n`-bit vector. If there is a majority element among the three input vectors, the module sets `maj-existsp` to `t` and returns the majority element. Otherwise, it sets `maj-existp` to `f` and returns an `n`-bit vector of `f`.

`Tv-if3` implements a nest of selectors (conditionals) that occurs several times in our implementation. The module is parameterized by `n`. It takes the inputs `c0`, `v0`, `c1`, `v1`, `c2`, `v2` and `v3`, where the `c`*i* are single signals and the `v`*i* are bit vectors of width `n`. Its output is the `n`-bit vector specified by `(if c0 v0 (if c1 v1 (if c2 v2 v3)))`.

`Regs3` is a parameterized state-holding module that consists of three `n`-bit registers. It takes as input three `n`-bit vectors (plus the usual `clk`, `te`, and `ti` used in all register modules), returns as its value the vectors in the three registers, and stores its input vectors as the new state of the registers. We use `regs3` to build a row of `local-step`'s matrix.

`Regs4` is like `regs3` except operates on four `n`-bit vectors. We use `regs4` to represent the `icv-reg` and the `act-reg`.

`v-buf-pwr` is a parameterized `n`-bit buffer module, a device that passes its `n` bits of input through but has more drive than a normal buffer. We use it in order to make our implementation acceptable to a certain formally defined predicate that checks the loads and drives on all our signals.

4.4 Lstep

We now describe our implementation of `local-step`. The module is called `lstep`. It is parameterized by `n`, the sense data width. See Appendix B for the definition of the module.

`Lstep` takes the input signals `clk`, `te`, `ti`, `reset-` and four `n`-bit input vectors, `sense`, `p0`, `p1` and `p2`. It is a state-holding module whose state `s` is satisfies `(hunt-brock-statep s n)`. It returns seven `n`-bit vectors, `o0`, `o1`, `o2`, `a0`, `a1`, `a2` and `a3`, and one 3-bit vector, `act-maj-existsp`. The three `o`*i* outputs represent `local-step`'s outputs to the three peer processors. The four `a`*i* outputs represent the ‘‘actuator icv’’—the four vectors determining the `light` or final action taken by the processor. The `act-maj-existsp` output indicates which of the first three `a`*i* actually denote `(maj-token)`.

The occurrences in the module are roughly described as follows. The three matrix rows are defined as instances of the `regs3` module. Matrix element `M02` is used so often we have to buffer it with `v-buf-pwr`. We define the `data-out` register as another instance of `regs3`, and take our three `n`-bit `o`*i* vectors from them. We define `icv-reg` as an instance of `regs4`. We define `act-reg` as an instance of `regs4` We define `icv-maj-existsp-reg` and the `act-maj-existsp-reg` each as instances of `reg3`.

We use `counter3` to obtain and increment the clock and then use `split-3-to-6` to demultiplex it into at most one ‘‘hot’’ signal. The six outputs are fanned out into the logic below so as to sequence the steps correctly. Two of the six, namely `s1` and `s2`, are used so often that we have to buffer them in order to drive all the dependent gates.

In the occurrences named `g1` through `g6-m22v`, we use `tv-if3` and the primitive `tv-if` to shuffle data between our inputs, `data-out` and the `matrix` rows as determined by which of the multiplexed clock signals is `t`.

In the occurrences named **g7**, **g8**, and **g9** we vote on the appropriate combinations of **matrix** elements, using **majority3** to obtain both the **maj-existsp** bit and the answer for each of the three votes. This is done on every cycle but the results are ignored except when the clock signal **s4** is **t**, when we put the results into **icv-reg** and **icv-maj-existsp-reg** (in occurrences **g11-icv0v** through **g11-icv2v** and **g13**). At occurrence **g11-icv3v** we put the **sense** input into **icv3** when the clock signal **s0** is **t**.

In occurrences **g12-a0v** through **g12-a3v** we load **act-reg** from **icv-reg** if the clock signal **s5** is **t**. At **g12-act-maj-exists** we load **act-maj-existsp-reg** from **icv-maj-existsp-reg** if the clock signal **s5** is **t**.

Because **lstep** is parameterized we cannot exhibit an NDL display of it. But we can exhibit the NDL for an instance. In Appendix C we show some of the NDL generated for the 8-bit wide version of **lstep**. In Appendix D we include the top-level schematic for that instance of **lstep**.

5. The Theorem Proved by NQTHM

We have proved the following theorem about **lstep**.

Theorem.
```
(implies
 (and (not (zerop w))
      (bevier-young-statep state act-reg w)
      (lstep& netlist w)
      (bvpn sense w)
      (bvpn p0 w)
      (bvpn p1 w)
      (bvpn p2 w)
      (equal te f)
      (equal reset- t))
 (equal (local-step (list sense p0 p1 p2)
                   state)
        (up (dual-eval '2 (index 'lstep w)
                       (cons clk
                        (cons te
                         (cons ti
                          (cons reset-
                                (append sense
                                        (append p0
                                                (append p1 p2)))))))
                       (down state act-reg w)
                       netlist))))
```

Observe the similarity between this theorem, proved by NQTHM, and the specification of the hardware, **Main**. In particular, if we let **w**, above, be **8** and **netlist**, above, be **(lstep$netlist 8)**, and we observe that **(not (zerop 8))** and **(lstep& (lstep$netlist 8) 8)**, then the indicated instance of the theorem above is just **Main**. Put another way, if we generate a netlist of the desired width with **lstep$netlist** and it passes the **lstep&** test (which can be determined by computation), then we know the netlist implements **local-step**.

Part of the NDL translation of **(lstep$netlist 8)** is shown in Appendix **NDL**.

It should be noted that the netlist produced by **(lstep$netlist 8)** passes the NQTHM predicate that checks adherence to various design rules, including those constraining the loads and drives in the net.

6. Comments on our Design

After obtaining NDL for our verified design, we used LSI Logic, Inc. tools to analyze the design. One such tool summarizes how our design uses the LSI gate array on which it could be built, the LMA9141C.

```
***********************************************************************
*                                                                     *
*               LDS-III DESIGN VERIFIER NETWORK SUMMARY               *
*                                                                     *
*  PROJECT ID:            L1A6477     LDS ACCOUNT NAME:      MDEACCT1  *
*  ARRAY NAME:            LSTEP_8     ARRAY FAMILY:          LMA9K     *
*  ARRAY TYPE:            LMA9141C                                     *
*                                                                     *
*  CURRENT DATE:          09/04/91    CURRENT TIME:          16:26:10  *
*  LMA9K LIBRARY DATE:    12/13/90    LMA9K LIBRARY REVISION:   10.12.0*
*  MEM10K LIBRARY DATE:   08/09/90    MEM10K LIBRARY REVISION:  10.09  *
*                                                                     *
***********************************************************************
*                                                                     *
*               NETWORK STATISTICS AFTER CELL DELETIONS               *
*                                                                     *
*                                                                     *
*  NUMBER OF CELLS DELETED: ..................................    0  *
*  NUMBER OF UNCONNECTED CELL OUTPUTS: ........................  244  *
*                                                                     *
*  NUMBER OF INPUT PINS (EXCLUDING BIDIRECTIONAL PINS): ... 36         *
*  NUMBER OF OUTPUT PINS (EXCLUDING BIDIRECTIONAL PINS): .. 59         *
*  NUMBER OF BIDIRECTIONAL PINS: ..........................  0         *
*  TOTAL NUMBER OF I/O SIGNAL PINS USED: ......................   95  *
*                                                                     *
*  RANGE OF POWER PINS REQUIRED (VSS & VDD) [min-max]: ........ 08-16  *
*                                                                     *
*  NUMBER OF PAD LOCATIONS USED FOR INPUT PINS: ..........  0         *
*  NUMBER OF PAD LOCATIONS USED FOR OUTPUT PINS: ..........  0         *
*  NUMBER OF PAD LOCATIONS USED FOR BIDIRECTIONAL PINS: ...  0         *
*  TOTAL NUMBER OF PAD LOCATIONS USED FOR ABOVE: ..............    0  *
*                                                                     *
*  TOTAL NUMBER OF UNRESERVED PAD LOCATIONS AVAILABLE: .........  110  *
*                                                                     *
*  NUMBER OF I/O DEVICE LOCATIONS USED FOR BUFFERS: ...........    0  *
*  TOTAL NUMBER OF I/O DEVICE LOCATIONS AVAILABLE: ............  114  *
*                                                                     *
*  NUMBER OF CELLS USED:       791    NUMBER OF GATES USED:      3438  *
*  NUMBER OF CELL TYPES:        12    ARRAY GATE USAGE (%):     24.34  *
*  MAXIMUM PINS PER NET:       169    ARRAY AREA USAGE (%):     24.34  *
*  NETS WITH 10<PINS/NET<=20:    3    NUMBER OF SIGNAL NETS:      826  *
*  NETS WITH PINS/NET > 20:      2    AVERAGE PINS PER NET:     3.442  *
*                                                                     *
***********************************************************************
```

Observe that our design has 3438 gates. The number of io pins is 95. This is excessively high. It is due to the fact that our design uses parallel io on 8-bit wide vectors. Recall that there are four 8-bit input vectors plus four single-bit signals, for a total of 36 input pins. The module has seven 8-bit output vectors plus three single-bit signals, for a total of 59 output pins. If one wished to exchange 32-bit wide sense data, the number of pins required would be 359! Our design is parameterized by the data size and our netlist generator produces correct designs for arbitrary data sizes. But such a summary is deceptive because the design is not practical for realistic data sizes.

A more sensible design would use serial io, devoting one pin to each of the channels on which full vectors are currently exchanged. This would reduce the pin count to eighteen and allow arbitrarily sized data at the cost of waiting for it to stream in. In [4], we verify that a biphase mark communications protocol allows reliable communication between two processors whose cycle times are within about 5% of each other. The reader of this document will recognize that it would be straightforward to implement the biphase mark specification in our Formal HDL and prove that we had done so. Proving that an HDL description implemented the **`send`** and **`recv`** of [4] would be an exercise very similar to proving that **`lstep`** implements **`local-step`**—except it would be easier because there is no need to parameterize the implementation and the state mapping is much simpler. Indeed, the whole approach taken in [4] was motivated by our concern that the verification of the implementation of **`send`** and **`recv`** be straightforward and independent of all extraneous considerations. The straightforward implementation of those two functions would allow data to be sent at the burst rate of 1.1M bps if we clocked the microprocessors at 20MHz and had a suitable channel between them.

Our **`lstep`**—even ignoring its excessive pin requirements— is not suitable for fault-tolerant applications because of the common clock assumption. Our processor implements **`local-step`**. **`Local-step`** was proved by Bevier and Young to provide fault-tolerance when it was connected in a network with three identical peers, all of which step in concert. More realistically, the four processors should each have an independent clock. An algorithm like that verified in [6] should be used to get the processors in approximate synchronization, so that they are all executing the same step of the algorithm during the same time interval. Our model of asynchronous communications [4] would permit us to prove that two such processors could communicate.

As we envision it, the low level specification of a realistic Byzantine agreement processor will be a function, say **`async-local-step`**, which is like **`local-step`** but has a much finer temporal grain. **`Async-local-step`** will break each of the six steps of **`local-step`** into hundreds cycles and allow for serial communication, clock synchronization, and a certain amount of waiting to keep each major step sufficiently large to insure that all processors step more or less together. Under an appropriate state mapping, which would necessarily include some time abstraction, **`async-local-step`** could be shown to implement **`local-step`**. The Formal HDL design would use **`async-local-step`**, not **`local-step`**, as the specification. We offer this sketch of a realistic design effort merely to emphasize how far we are from having achieved it.

Appendix A. The Formal Definition of LOCAL-STEP and GOOD-STATEP

Definition .

```
(length l)
   =
(if (listp l)
    (add1 (length (cdr l)))
    0)
```

Shell Definition.
Add the **`state`** of five arguments
with recognizer **`statep`** and
accessors **`matrix`**, **`obuf`**, **`icv`**, **`light`** and **`clock`**.

Definition .

```
(make-list length initial-value)
   =
(if (zerop length)
    nil
    (cons initial-value
          (make-list (sub1 length) initial-value)))
```

Definition .

```
(nth n l)
   =
(if (listp l)
    (if (zerop n)
        (car l)
        (nth (sub1 n) (cdr l)))
    0)
```

Definition.

```
(put n v l)
   =
(if (listp l)
    (if (zerop n)
        (cons v (cdr l))
        (cons (car l) (put (sub1 n) v (cdr l))))
    l)
```

Definition.

```
(nth2 i j x)
   =
(nth j (nth i x))
```

In the original Bevier-Young work, **`MAJORITY`** was introduced by constraint. However, the function was constrained to the point of being uniquely defined. We simply define it and its companion, **`MAJORITY-EXISTS`**. In our mechanical proof script we include the events that establish that our functions satisfy the constraints imposed on theirs. The uniqueness of their functions is not proved in our script, though we have (elsewhere) led NQTHM to that conclusion.

Definition.

```
(occurrences
      x l)
   =
(if (listp l)
    (if (equal x (car l))
        (add1 (occurrences x (cdr l)))
        (occurrences x (cdr l)))
    0)
```

Definition.

```
(majority1 cands votes)
   =
(if (listp cands)
    (if (lessp (length votes)
               (times 2 (occurrences (car cands) votes)))
        (car cands)
        (majority1 (cdr cands) votes))
    0)
```

Definition.

```
(majority-exists1 cands votes)
   =
(if (listp cands)
    (or (lessp (length votes) (times 2 (occurrences (car cands) votes)))
        (majority-exists1 (cdr cands) votes))
    f)
```

Shell Definition.
Add the shell **`maj-token`**
with recognizer **`maj-tokenp`**.

Definition.

```
(majority votes)
   =
(if (majority-exists1 votes votes)
    (majority1 votes votes)
    (maj-token))
```

Definition.

```
(majority-exists votes)
   =
(majority-exists1 votes votes)
```

Definition.

```
(compute-icv matrix icv)
   =
(put 0
     (majority (list (nth2 0 0 matrix)
                     (nth2 1 2 matrix)
                     (nth2 2 1 matrix)))
     (put 1
          (majority (list (nth2 0 1 matrix)
                          (nth2 1 0 matrix)
                          (nth2 2 2 matrix)))
          (put 2
               (majority (list (nth2 0 2 matrix)
                               (nth2 1 1 matrix)
                               (nth2 2 0 matrix)))
               icv)))
```

The Bevier-Young function `filter` was introduced by constraint. We do not need any properties of `filter` and thus introduce it by declaration (i.e., as an undefined, unconstrained function symbol).

Undefined Function.
```
(filter icv)
```

Definition.
```
(tablep n l)
   =
(if (listp l)
    (and (equal (length (car l)) (fix n))
         (tablep n (cdr l)))
    t)
```

Definition.
```
(matrixp i j l)
   =
(and (equal (length l) (fix i))
     (tablep j l))
```

Definition.
```
(good-statep x)
   =
(and (statep x)
     (matrixp 3 3 (matrix x))
     (equal (length (obuf x)) 3)
     (equal (length (icv x)) 4)
     (numberp (clock x))
     (lessp (clock x) 8))
```

Definition.
```
(local-step input state)
   =
(if (equal (remainder (clock state) 8) 0)
    (state (matrix state)
           (make-list 3 (nth 0 input))
           (put 3 (nth 0 input) (icv state))
           (light state)
           (remainder (plus 1 (clock state)) 8))
(if (equal (remainder (clock state) 8) 1)
    (state (put 0
                (list (nth 1 input)
                      (nth 2 input)
                      (nth 3 input))
                (matrix state))
           (list (nth 2 input)
                 (nth 1 input)
                 (nth 1 input))
           (icv state)
           (light state)
           (remainder (plus 1 (clock state)) 8))
(if (equal (remainder (clock state) 8) 2)
    (state (put 1
                (list (nth 1 input)
                      (nth 2 input)
                      (nth 3 input))
                (matrix state))
           (list (nth 2 (nth 0 (matrix state)))
                 (nth 2 (nth 0 (matrix state)))
                 (nth 1 (nth 0 (matrix state))))
           (icv state)
           (light state)
           (remainder (plus 1 (clock state)) 8))
(if (equal (remainder (clock state) 8) 3)
    (state (put 2
                (list (nth 1 input)
                      (nth 2 input)
                      (nth 3 input) nil)
                (matrix state))
           (obuf state)
           (icv state)
           (light state)
           (remainder (plus 1 (clock state)) 8))
(if (equal (remainder (clock state) 8) 4)
    (state (matrix state)
           (obuf state)
           (compute-icv (matrix state) (icv state))
           (light state)
           (remainder (plus 1 (clock state)) 8))
(if (equal (remainder (clock state) 8) 5)
    (state (matrix state)
           (obuf state)
           (icv state)
           (filter (icv state))
           (remainder (plus 1 (clock state)) 8))
    (state (matrix state)
           (obuf state)
           (icv state)
           (light state)
           (remainder (plus 1 (clock state)) 8))))))))
```

Appendix B. The Formal Design

We exhibit the functions that generate each our modules. For each such generator, *fn*`*`, there is also a netlist generator *fn*`$NETLIST` and a netlist recognizer *fn*`&`. The netlist generator returns a list of the generated module and each of its submodules. The netlist recognizer checks that the given netlist contains the generated module and each of the required submodules.

Definition.

```
(INCRMT3*)
   =
'(INCRMT3 (I0 I1 I2)
          (O0 O1 O2)
          ((G0 (O0) B-NOT (I0))
           (G1 (O1) B-XOR (I0 I1))
           (G2 (S0) B-AND (I0 I1))
           (G3 (O2) B-XOR (S0 I2)))
          NIL)
```

Definition.

```
(COUNTER3-TEMP*)
   =
'(COUNTER3-TEMP (RESET- I0 I1 I2)
                (D0 D1 D2)
                ((G0 (D0) B-AND (RESET- I0))
                 (G1 (D1) B-AND (RESET- I1))
                 (G2 (D2) B-AND (RESET- I2)))
                NIL)
```

Definition.

```
(COUNTER3*)
   =
(LIST ’COUNTER3
      ’(CLK TE TI RESET-)
      (INDICES ’Q 0 3)
      (LIST (LIST ’REG
                  (INDICES ’Q 0 3)
                  (INDEX ’REG 3)
                  (CONS ’CLK
                        (CONS ’TE
                              (CONS ’TI (INDICES ’D 0 3)))))
            (LIST ’INC
                  (INDICES ’I 0 3)
                  ’INCRMT3
                  (INDICES ’Q 0 3))
            (LIST ’G0
                  (INDICES ’D 0 3)
                  ’COUNTER3-TEMP
                  (CONS ’RESET- (INDICES ’I 0 3))))
      ’(REG))
```

Definition.

```
(SPLIT-3-TO-6*)
   =
’(SPLIT-3-TO-6 (C0 C1 C2)
               (S0 S1 S2 S3 S4 S5)
               ((G0 (NC0) B-NOT (C0))
                (G1 (NC1) B-NOT (C1))
                (G2 (NC2) B-NOT (C2))
                (G3 (S0) B-AND3 (NC0 NC1 NC2))
                (G4 (S1) B-AND3 (C0 NC1 NC2))
                (G5 (S2) B-AND3 (NC0 C1 NC2))
                (G6 (S3) B-AND3 (C0 C1 NC2))
                (G7 (S4) B-AND3 (NC0 NC1 C2))
                (G8 (S5) B-AND3 (C0 NC1 C2)))
               NIL)
```

Definition.

```
(MAJORITY3* N)
   =
(LIST (INDEX ’MAJORITY3 N)
      (APPEND (INDICES ’X 0 N)
              (APPEND (INDICES ’Y 0 N)
                      (INDICES ’Z 0 N)))
      (CONS ’MAJ-EXISTSP (INDICES ’A 0 N))
      (LIST (LIST ’G0 ’(E0)
                  (INDEX ’V-EQUAL N)
                  (APPEND (INDICES ’X 0 N)
                          (INDICES ’Y 0 N)))
            (LIST ’G1 ’(E1)
                  (INDEX ’V-EQUAL N)
                  (APPEND (INDICES ’X 0 N)
                          (INDICES ’Z 0 N)))
            (LIST ’G2 ’(E2)
                  (INDEX ’V-EQUAL N)
                  (APPEND (INDICES ’Y 0 N)
                          (INDICES ’Z 0 N)))
            (LIST ’G2A
                  (INDICES ’ZERO 0 N)
                  (INDEX ’V-XOR N)
                  (APPEND (INDICES ’X 0 N)
                          (INDICES ’X 0 N)))
            ’(G3 (MAJ-EXISTSP) B-OR3 (E0 E1 E2))
            (LIST ’G4 (INDICES ’C 0 N)
                  (INDEX ’TV-IF
                         (TREE-NUMBER (MAKE-TREE N)))
                  (CONS ’E2
                        (APPEND (INDICES ’Y 0 N)
                                (INDICES ’ZERO 0 N))))
            (LIST ’G5 (INDICES ’B 0 N)
                  (INDEX ’TV-IF
                         (TREE-NUMBER (MAKE-TREE N)))
                  (CONS ’E1
                        (APPEND (INDICES ’X 0 N)
                                (INDICES ’C 0 N))))
            (LIST ’G6 (INDICES ’A 0 N)
                  (INDEX ’TV-IF
                         (TREE-NUMBER (MAKE-TREE N)))
                  (CONS ’E0
                        (APPEND (INDICES ’X 0 N)
                                (INDICES ’B 0 N)))))
      NIL)
```

Definition.

```
(TV-IF3* N)
   =
(LIST (INDEX ’TV-IF3 N)
      (CONS ’C0
            (APPEND (INDICES ’V0 0 N)
                    (CONS ’C1
                          (APPEND (INDICES ’V1 0 N)
                                  (CONS ’C2
                                        (APPEND (INDICES ’V2 0 N)
                                                (INDICES ’V3 0 N)))))))
      (INDICES ’OUTPUT 0 N)
      (LIST (LIST ’G0 (INDICES ’T1 0 N)
                  (INDEX ’TV-IF
                         (TREE-NUMBER (MAKE-TREE N)))
                  (CONS ’C2
                        (APPEND (INDICES ’V2 0 N)
                                (INDICES ’V3 0 N))))
            (LIST ’G1 (INDICES ’T2 0 N)
                  (INDEX ’TV-IF
                         (TREE-NUMBER (MAKE-TREE N)))
                  (CONS ’C1
                        (APPEND (INDICES ’V1 0 N)
                                (INDICES ’T1 0 N))))
            (LIST ’G2 (INDICES ’OUTPUT 0 N)
                  (INDEX ’TV-IF
                         (TREE-NUMBER (MAKE-TREE N)))
                  (CONS ’C0
                        (APPEND (INDICES ’V0 0 N)
                                (INDICES ’T2 0 N)))))
      NIL)
```

Definition.

```
(REGS3* N)
   =
(LIST (INDEX ’REGS3 N)
      (CONS ’CLK
            (CONS ’TE
                  (CONS ’TI
                        (APPEND (INDICES ’R0 0 N)
                                (APPEND (INDICES ’R1 0 N)
                                        (INDICES ’R2 0 N))))))
      (APPEND (INDICES ’Q0 0 N)
              (APPEND (INDICES ’Q1 0 N)
                      (INDICES ’Q2 0 N)))
      (LIST (LIST ’REG0 (INDICES ’Q0 0 N)
                  (INDEX ’REG N)
                  (CONS ’CLK
                        (CONS ’TE
                              (CONS ’TI (INDICES ’R0 0 N)))))
            (LIST ’REG1 (INDICES ’Q1 0 N)
                  (INDEX ’REG N)
                  (CONS ’CLK
                        (CONS ’TE
                              (CONS (INDEX ’Q0 (SUB1 N))
                                    (INDICES ’R1 0 N)))))
            (LIST ’REG2 (INDICES ’Q2 0 N)
                  (INDEX ’REG N)
                  (CONS ’CLK
                        (CONS ’TE
                              (CONS (INDEX ’Q1 (SUB1 N))
                                    (INDICES ’R2 0 N))))))
      ’(REG0 REG1 REG2))
```

Definition.

```
(REGS4* N)
   =
(LIST (INDEX ’REGS4 N)
      (CONS ’CLK
            (CONS ’TE
                  (CONS ’TI
                        (APPEND (INDICES ’R0 0 N)
                                (APPEND (INDICES ’R1 0 N)
                                        (APPEND (INDICES ’R2 0 N)
                                                (INDICES ’R3 0 N)))))))
      (APPEND (INDICES ’Q0 0 N)
              (APPEND (INDICES ’Q1 0 N)
                      (APPEND (INDICES ’Q2 0 N)
                              (INDICES ’Q3 0 N))))
      (LIST (LIST ’REG0 (INDICES ’Q0 0 N)
                  (INDEX ’REG N)
                  (CONS ’CLK
                        (CONS ’TE
                              (CONS ’TI (INDICES ’R0 0 N)))))
            (LIST ’REG1 (INDICES ’Q1 0 N)
                  (INDEX ’REG N)
                  (CONS ’CLK
                        (CONS ’TE
                              (CONS (INDEX ’Q0 (SUB1 N))
                                    (INDICES ’R1 0 N)))))
            (LIST ’REG2 (INDICES ’Q2 0 N)
                  (INDEX ’REG N)
                  (CONS ’CLK
                        (CONS ’TE
                              (CONS (INDEX ’Q1 (SUB1 N))
                                    (INDICES ’R2 0 N)))))
            (LIST ’REG3 (INDICES ’Q3 0 N)
                  (INDEX ’REG N)
                  (CONS ’CLK
                        (CONS ’TE
                              (CONS (INDEX ’Q2 (SUB1 N))
                                    (INDICES ’R3 0 N))))))
      ’(REG0 REG1 REG2 REG3))
```

Definition.

```
(V-BUF-PWR$BODY M N)
   =
(IF (ZEROP N)
    NIL
    (CONS (LIST (INDEX ’G M)
                (LIST (INDEX ’Y M))
                ’B-BUF-PWR
                (LIST (INDEX ’A M)))
          (V-BUF-PWR$BODY (ADD1 M) (SUB1 N))))
```

Definition.

```
(V-BUF-PWR* N)
   =
(LIST (INDEX ’V-BUF-PWR N)
      (INDICES ’A 0 N)
      (INDICES ’Y 0 N)
      (V-BUF-PWR$BODY 0 N)
      NIL)
```

Definition.

```
(LSTEP* N)
   =
(LIST
 (INDEX 'LSTEP N)
 (CONS 'CLK
       (CONS 'TE
             (CONS 'TI
                   (CONS 'RESET-
                         (APPEND (INDICES 'SENSE 0 N)
                                 (APPEND (INDICES 'P0 0 N)
                                         (APPEND (INDICES 'P1 0 N)
                                                 (INDICES 'P2 0 N))))))))
 (APPEND
  (INDICES 'O0 0 N)
  (APPEND (INDICES 'O1 0 N)
          (APPEND (INDICES 'O2 0 N)
                  (APPEND (INDICES 'ACT-MAJ-EXISTSP 0 3)
                          (APPEND (INDICES 'A0 0 N)
                                  (APPEND (INDICES 'A1 0 N)
                                          (APPEND (INDICES 'A2 0 N)
                                                  (INDICES 'A3 0 N))))))))
 (LIST
  '(CNT (C0 C1 C2)
        COUNTER3
        (CLK TE TI RESET-))
  '(G0 (S0 S1WEAK S2WEAK S3 S4 S5)
       SPLIT-3-TO-6
       (C0 C1 C2))
  '(G0A (S1) B-BUF-PWR (S1WEAK))
  '(G0B (S2) B-BUF-PWR (S2WEAK))
  (LIST 'MATRIX0
        (APPEND (INDICES 'M00 0 N)
                (APPEND (INDICES 'M01 0 N)
                        (INDICES 'M02WEAK 0 N)))
        (INDEX 'REGS3 N)
        (CONS 'CLK
              (CONS 'TE
                    (CONS 'C2
                          (APPEND (INDICES 'M00V 0 N)
                                  (APPEND (INDICES 'M01V 0 N)
                                          (INDICES 'M02V 0 N)))))))
  (LIST 'MATRIX0A
        (INDICES 'M02 0 N)
        (INDEX 'V-BUF-PWR N)
        (INDICES 'M02WEAK 0 N))
  (LIST 'MATRIX1
        (APPEND (INDICES 'M10 0 N)
                (APPEND (INDICES 'M11 0 N)
                        (INDICES 'M12 0 N)))
        (INDEX 'REGS3 N)
        (CONS 'CLK
              (CONS 'TE
                    (CONS (INDEX 'M02 (SUB1 N))
                          (APPEND (INDICES 'M10V 0 N)
                                  (APPEND (INDICES 'M11V 0 N)
                                          (INDICES 'M12V 0 N)))))))
  (LIST 'MATRIX2
        (APPEND (INDICES 'M20 0 N)
                (APPEND (INDICES 'M21 0 N)
                        (INDICES 'M22 0 N)))
        (INDEX 'REGS3 N)
        (CONS 'CLK
              (CONS 'TE
                    (CONS (INDEX 'M12 (SUB1 N))
                          (APPEND (INDICES 'M20V 0 N)
                                  (APPEND (INDICES 'M21V 0 N)
                                          (INDICES 'M22V 0 N)))))))
  (LIST 'DATA-OUT
        (APPEND (INDICES 'O0 0 N)
                (APPEND (INDICES ’O1 0 N)
                        (INDICES ’O2 0 N)))
        (INDEX ’REGS3 N)
        (CONS ’CLK
              (CONS ’TE
                    (CONS (INDEX ’M22 (SUB1 N))
                          (APPEND (INDICES ’O0V 0 N)
                                  (APPEND (INDICES ’O1V 0 N)
                                          (INDICES ’O2V 0 N)))))))
  (LIST ’ICV-REG
        (APPEND (INDICES ’ICV0 0 N)
                (APPEND (INDICES ’ICV1 0 N)
                        (APPEND (INDICES ’ICV2 0 N)
                                (INDICES ’ICV3 0 N))))
        (INDEX ’REGS4 N)
        (CONS ’CLK
              (CONS ’TE
                    (CONS (INDEX ’O2 (SUB1 N))
                          (APPEND (INDICES ’ICV0V 0 N)
                                  (APPEND (INDICES ’ICV1V 0 N)
                                          (APPEND (INDICES ’ICV2V 0 N)
                                                  (INDICES ’ICV3V 0 N))))))))
  (LIST ’ACT-REG
        (APPEND (INDICES ’A0 0 N)
                (APPEND (INDICES ’A1 0 N)
                        (APPEND (INDICES ’A2 0 N)
                                (INDICES ’A3 0 N))))
        (INDEX ’REGS4 N)
        (CONS ’CLK
              (CONS ’TE
                    (CONS (INDEX ’ICV3 (SUB1 N))
                          (APPEND (INDICES ’A0V 0 N)
                                  (APPEND (INDICES ’A1V 0 N)
                                          (APPEND (INDICES ’A2V 0 N)
                                                  (INDICES ’A3V 0 N))))))))
  (LIST ’ICV-MAJ-EXISTSP-REG
        (INDICES ’ICV-MAJ-EXISTSP 0 3)
        (INDEX ’REG 3)
        (CONS ’CLK
              (CONS ’TE
                    (CONS (INDEX ’A3 (SUB1 N))
                          (INDICES ’ICV-MAJ-EXISTSPV 0 3)))))
  (LIST ’ACT-MAJ-EXISTSP-REG
        (INDICES ’ACT-MAJ-EXISTSP 0 3)
        (INDEX ’REG 3)
        (CONS ’CLK
              (CONS ’TE
                    (CONS (INDEX ’ICV-MAJ-EXISTSP 2)
                          (INDICES ’ACT-MAJ-EXISTSPV 0 3)))))
  (LIST ’G1
        (INDICES ’O0V 0 N)
        (INDEX ’TV-IF3 N)
        (CONS ’S0
              (APPEND (INDICES ’SENSE 0 N)
                      (CONS ’S1
                            (APPEND (INDICES ’P1 0 N)
                                    (CONS ’S2
                                          (APPEND (INDICES ’M02 0 N)
                                                  (INDICES ’O0 0 N))))))))
  (LIST ’G2
        (INDICES ’O1V 0 N)
        (INDEX ’TV-IF3 N)
        (CONS ’S0
              (APPEND (INDICES ’SENSE 0 N)
                      (CONS ’S1
                            (APPEND (INDICES ’P0 0 N)
                                    (CONS ’S2
                                          (APPEND (INDICES ’M02 0 N)
                                                  (INDICES ’O1 0 N))))))))
```

```
(LIST ’G3
      (INDICES ’O2V 0 N)
      (INDEX ’TV-IF3 N)
      (CONS ’S0
            (APPEND (INDICES ’SENSE 0 N)
                    (CONS ’S1
                          (APPEND (INDICES ’P0 0 N)
                                  (CONS ’S2
                                        (APPEND (INDICES ’M01 0 N)
                                                (INDICES ’O2 0 N))))))))
(LIST ’G4-M00V
      (INDICES ’M00V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S1
            (APPEND (INDICES ’P0 0 N)
                    (INDICES ’M00 0 N))))
(LIST ’G4-M01V
      (INDICES ’M01V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S1
            (APPEND (INDICES ’P1 0 N)
                    (INDICES ’M01 0 N))))
(LIST ’G4-M02V
      (INDICES ’M02V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S1
            (APPEND (INDICES ’P2 0 N)
                    (INDICES ’M02 0 N))))
(LIST ’G5-M10V
      (INDICES ’M10V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S2
            (APPEND (INDICES ’P0 0 N)
                    (INDICES ’M10 0 N))))
(LIST ’G5-M11V
      (INDICES ’M11V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S2
            (APPEND (INDICES ’P1 0 N)
                    (INDICES ’M11 0 N))))
(LIST ’G5-M12V
      (INDICES ’M12V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S2
            (APPEND (INDICES ’P2 0 N)
                    (INDICES ’M12 0 N))))
(LIST ’G6-M20V
      (INDICES ’M20V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S3
            (APPEND (INDICES ’P0 0 N)
                    (INDICES ’M20 0 N))))
(LIST ’G6-M21V
      (INDICES ’M21V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S3
            (APPEND (INDICES ’P1 0 N)
                    (INDICES ’M21 0 N))))
(LIST ’G6-M22V
      (INDICES ’M22V 0 N)
      (INDEX ’TV-IF
              (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S3
            (APPEND (INDICES ’P2 0 N)
                    (INDICES ’M22 0 N))))
(LIST ’G7
      (CONS (INDEX ’ICV-MAJ-EXISTSPV1 0)
            (INDICES ’MJRTY0 0 N))
      (INDEX ’MAJORITY3 N)
      (APPEND (INDICES ’M00 0 N)
              (APPEND (INDICES ’M12 0 N)
                      (INDICES ’M21 0 N))))
(LIST ’G8
      (CONS (INDEX ’ICV-MAJ-EXISTSPV1 1)
            (INDICES ’MJRTY1 0 N))
      (INDEX ’MAJORITY3 N)
      (APPEND (INDICES ’M01 0 N)
              (APPEND (INDICES ’M10 0 N)
                      (INDICES ’M22 0 N))))
(LIST ’G9
      (CONS (INDEX ’ICV-MAJ-EXISTSPV1 2)
            (INDICES ’MJRTY2 0 N))
      (INDEX ’MAJORITY3 N)
      (APPEND (INDICES ’M02 0 N)
              (APPEND (INDICES ’M11 0 N)
                      (INDICES ’M20 0 N))))
(LIST ’G11-ICV0V
      (INDICES ’ICV0V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S4
            (APPEND (INDICES ’MJRTY0 0 N)
                    (INDICES ’ICV0 0 N))))
(LIST ’G11-ICV1V
      (INDICES ’ICV1V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S4
            (APPEND (INDICES ’MJRTY1 0 N)
                    (INDICES ’ICV1 0 N))))
(LIST ’G11-ICV2V
      (INDICES ’ICV2V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S4
            (APPEND (INDICES ’MJRTY2 0 N)
                    (INDICES ’ICV2 0 N))))
(LIST ’G11-ICV3V
      (INDICES ’ICV3V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S0
            (APPEND (INDICES ’SENSE 0 N)
                    (INDICES ’ICV3 0 N))))
(LIST ’G12-A0V
      (INDICES ’A0V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S5
            (APPEND (INDICES ’ICV0 0 N)
                    (INDICES ’A0 0 N))))
(LIST ’G12-A1V
      (INDICES ’A1V 0 N)
      (INDEX ’TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS ’S5
            (APPEND (INDICES ’ICV1 0 N)
                    (INDICES ’A1 0 N))))
(LIST ’G12-A2V
      (INDICES ’A2V 0 N)
      (INDEX 'TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS 'S5
            (APPEND (INDICES 'ICV2 0 N)
                    (INDICES 'A2 0 N))))
(LIST 'G12-A3V
      (INDICES 'A3V 0 N)
      (INDEX 'TV-IF
             (TREE-NUMBER (MAKE-TREE N)))
      (CONS 'S5
            (APPEND (INDICES 'ICV3 0 N)
                    (INDICES 'A3 0 N))))
(LIST 'G12-ACT-MAJ-EXISTS
      (INDICES 'ACT-MAJ-EXISTSPV 0 3)
      (INDEX 'TV-IF
             (TREE-NUMBER (MAKE-TREE 3)))
      (CONS 'S5
            (APPEND (INDICES 'ICV-MAJ-EXISTSP 0 3)
                    (INDICES 'ACT-MAJ-EXISTSP 0 3))))
(LIST 'G13
      (INDICES 'ICV-MAJ-EXISTSPV 0 3)
      (INDEX 'TV-IF
             (TREE-NUMBER (MAKE-TREE 3)))
      (CONS 'S4
            (APPEND (INDICES 'ICV-MAJ-EXISTSPV1 0 3)
                    (INDICES 'ICV-MAJ-EXISTSP 0 3)))))
'(CNT MATRIX0 MATRIX1 MATRIX2 DATA-OUT ICV-REG ACT-REG
     ICV-MAJ-EXISTSP-REG ACT-MAJ-EXISTSP-REG))
```

Appendix C. The NDL for LSTEP_8

Below we display part of a netlist that has been proved (by construction) to implement **`local-step`** for a data path width of 8. The syntax of the display is NDL, the Netlist Description Language of LSI Logic Inc. The complete netlist occupies about 9 pages.

```
MODULE LSTEP_8;
INPUTS CLK,TE,TI,RESET-,SENSE.0,SENSE.1,SENSE.2,SENSE.3,SENSE.4,SENSE.5,
       SENSE.6,SENSE.7,P0.0,P0.1,P0.2,P0.3,P0.4,P0.5,P0.6,P0.7,P1.0,P1.1,
       P1.2,P1.3,P1.4,P1.5,P1.6,P1.7,P2.0,P2.1,P2.2,P2.3,P2.4,P2.5,P2.6,P2.7;
OUTPUTS O0.0,O0.1,O0.2,O0.3,O0.4,O0.5,O0.6,O0.7,O1.0,O1.1,O1.2,O1.3,O1.4,
        O1.5,O1.6,O1.7,O2.0,O2.1,O2.2,O2.3,O2.4,O2.5,O2.6,O2.7,
        ACT-MAJ-EXISTSP.0,ACT-MAJ-EXISTSP.1,ACT-MAJ-EXISTSP.2,A0.0,A0.1,A0.2,
        A0.3,A0.4,A0.5,A0.6,A0.7,A1.0,A1.1,A1.2,A1.3,A1.4,A1.5,A1.6,A1.7,
        A2.0,A2.1,A2.2,A2.3,A2.4,A2.5,A2.6,A2.7,A3.0,A3.1,A3.2,A3.3,A3.4,
        A3.5,A3.6,A3.7;
LEVEL FUNCTION;
DEFINE
CNT(C0,C1,C2) = COUNTER3(CLK,TE,TI,RESET-);
G0(S0,S1WEAK,S2WEAK,S3,S4,S5) = SPLIT-3-TO-6(C0,C1,C2);
G0A(S1) = B-BUF-PWR(S1WEAK);
G0B(S2) = B-BUF-PWR(S2WEAK);
MATRIX0(M00.0,M00.1,M00.2,M00.3,M00.4,M00.5,M00.6,M00.7,M01.0,M01.1,M01.2,
        M01.3,M01.4,M01.5,M01.6,M01.7,M02WEAK.0,M02WEAK.1,M02WEAK.2,
        M02WEAK.3,M02WEAK.4,M02WEAK.5,M02WEAK.6,M02WEAK.7)
  = REGS3_8(CLK,TE,C2,M00V.0,M00V.1,M00V.2,M00V.3,M00V.4,M00V.5,M00V.6,
            M00V.7,M01V.0,M01V.1,M01V.2,M01V.3,M01V.4,M01V.5,M01V.6,M01V.7,
            M02V.0,M02V.1,M02V.2,M02V.3,M02V.4,M02V.5,M02V.6,M02V.7);
MATRIX0A(M02.0,M02.1,M02.2,M02.3,M02.4,M02.5,M02.6,M02.7)
  = V-BUF-PWR_8(M02WEAK.0,M02WEAK.1,M02WEAK.2,M02WEAK.3,M02WEAK.4,M02WEAK.5,
                M02WEAK.6,M02WEAK.7);
MATRIX1(M10.0,M10.1,M10.2,M10.3,M10.4,M10.5,M10.6,M10.7,M11.0,M11.1,M11.2,
        M11.3,M11.4,M11.5,M11.6,M11.7,M12.0,M12.1,M12.2,M12.3,M12.4,M12.5,
        M12.6,M12.7)
  = REGS3_8(CLK,TE,M02.7,M10V.0,M10V.1,M10V.2,M10V.3,M10V.4,M10V.5,M10V.6,
            M10V.7,M11V.0,M11V.1,M11V.2,M11V.3,M11V.4,M11V.5,M11V.6,M11V.7,
            M12V.0,M12V.1,M12V.2,M12V.3,M12V.4,M12V.5,M12V.6,M12V.7);
```

```
MATRIX2(M20.0,M20.1,M20.2,M20.3,M20.4,M20.5,M20.6,M20.7,M21.0,M21.1,M21.2,
        M21.3,M21.4,M21.5,M21.6,M21.7,M22.0,M22.1,M22.2,M22.3,M22.4,M22.5,
        M22.6,M22.7)
  = REGS3_8(CLK,TE,M12.7,M20V.0,M20V.1,M20V.2,M20V.3,M20V.4,M20V.5,M20V.6,
            M20V.7,M21V.0,M21V.1,M21V.2,M21V.3,M21V.4,M21V.5,M21V.6,M21V.7,
            M22V.0,M22V.1,M22V.2,M22V.3,M22V.4,M22V.5,M22V.6,M22V.7);
DATA-OUT(O0.0,O0.1,O0.2,O0.3,O0.4,O0.5,O0.6,O0.7,O1.0,O1.1,O1.2,O1.3,O1.4,
         O1.5,O1.6,O1.7,O2.0,O2.1,O2.2,O2.3,O2.4,O2.5,O2.6,O2.7)
  = REGS3_8(CLK,TE,M22.7,O0V.0,O0V.1,O0V.2,O0V.3,O0V.4,O0V.5,O0V.6,O0V.7,
            O1V.0,O1V.1,O1V.2,O1V.3,O1V.4,O1V.5,O1V.6,O1V.7,O2V.0,O2V.1,
            O2V.2,O2V.3,O2V.4,O2V.5,O2V.6,O2V.7);
ICV-REG(ICV0.0,ICV0.1,ICV0.2,ICV0.3,ICV0.4,ICV0.5,ICV0.6,ICV0.7,ICV1.0,
        ICV1.1,ICV1.2,ICV1.3,ICV1.4,ICV1.5,ICV1.6,ICV1.7,ICV2.0,ICV2.1,
        ICV2.2,ICV2.3,ICV2.4,ICV2.5,ICV2.6,ICV2.7,ICV3.0,ICV3.1,ICV3.2,
        ICV3.3,ICV3.4,ICV3.5,ICV3.6,ICV3.7)
  = REGS4_8(CLK,TE,O2.7,ICV0V.0,ICV0V.1,ICV0V.2,ICV0V.3,ICV0V.4,ICV0V.5,
            ICV0V.6,ICV0V.7,ICV1V.0,ICV1V.1,ICV1V.2,ICV1V.3,ICV1V.4,ICV1V.5,
            ICV1V.6,ICV1V.7,ICV2V.0,ICV2V.1,ICV2V.2,ICV2V.3,ICV2V.4,ICV2V.5,
            ICV2V.6,ICV2V.7,ICV3V.0,ICV3V.1,ICV3V.2,ICV3V.3,ICV3V.4,ICV3V.5,
            ICV3V.6,ICV3V.7);
...
G9(ICV-MAJ-EXISTSPV1.2,MJRTY2.0,MJRTY2.1,MJRTY2.2,MJRTY2.3,MJRTY2.4,MJRTY2.5,
   MJRTY2.6,MJRTY2.7)
  = MAJORITY3_8(M02.0,M02.1,M02.2,M02.3,M02.4,M02.5,M02.6,M02.7,M11.0,M11.1,
                M11.2,M11.3,M11.4,M11.5,M11.6,M11.7,M20.0,M20.1,M20.2,M20.3,
                M20.4,M20.5,M20.6,M20.7);
...
G11-ICV2V(ICV2V.0,ICV2V.1,ICV2V.2,ICV2V.3,ICV2V.4,ICV2V.5,ICV2V.6,ICV2V.7)
  = TV-IF_8(S4,MJRTY2.0,MJRTY2.1,MJRTY2.2,MJRTY2.3,MJRTY2.4,MJRTY2.5,
            MJRTY2.6,MJRTY2.7,ICV2.0,ICV2.1,ICV2.2,ICV2.3,ICV2.4,ICV2.5,
            ICV2.6,ICV2.7);
G11-ICV3V(ICV3V.0,ICV3V.1,ICV3V.2,ICV3V.3,ICV3V.4,ICV3V.5,ICV3V.6,ICV3V.7)
  = TV-IF_8(S0,SENSE.0,SENSE.1,SENSE.2,SENSE.3,SENSE.4,SENSE.5,SENSE.6,
            SENSE.7,ICV3.0,ICV3.1,ICV3.2,ICV3.3,ICV3.4,ICV3.5,ICV3.6,ICV3.7);
...
G12-A3V(A3V.0,A3V.1,A3V.2,A3V.3,A3V.4,A3V.5,A3V.6,A3V.7)
  = TV-IF_8(S5,ICV3.0,ICV3.1,ICV3.2,ICV3.3,ICV3.4,ICV3.5,ICV3.6,ICV3.7,A3.0,
            A3.1,A3.2,A3.3,A3.4,A3.5,A3.6,A3.7);
G12-ACT-MAJ-EXISTS(ACT-MAJ-EXISTSPV.0,ACT-MAJ-EXISTSPV.1,ACT-MAJ-EXISTSPV.2)
  = TV-IF_14(S5,ICV-MAJ-EXISTSP.0,ICV-MAJ-EXISTSP.1,ICV-MAJ-EXISTSP.2,
             ACT-MAJ-EXISTSP.0,ACT-MAJ-EXISTSP.1,ACT-MAJ-EXISTSP.2);
G13(ICV-MAJ-EXISTSPV.0,ICV-MAJ-EXISTSPV.1,ICV-MAJ-EXISTSPV.2)
  = TV-IF_14(S4,ICV-MAJ-EXISTSPV1.0,ICV-MAJ-EXISTSPV1.1,ICV-MAJ-EXISTSPV1.2,
             ICV-MAJ-EXISTSP.0,ICV-MAJ-EXISTSP.1,ICV-MAJ-EXISTSP.2);
END MODULE;

MODULE SPLIT-3-TO-6;
INPUTS C0,C1,C2;
OUTPUTS S0,S1,S2,S3,S4,S5;
LEVEL FUNCTION;
DEFINE
G0(NC0) = IVA(C0);
G1(NC1) = IVA(C1);
G2(NC2) = IVA(C2);
G3(S0) = AN3(NC0,NC1,NC2);
G4(S1) = AN3(C0,NC1,NC2);
G5(S2) = AN3(NC0,C1,NC2);
G6(S3) = AN3(C0,C1,NC2);
G7(S4) = AN3(NC0,NC1,C2);
G8(S5) = AN3(C0,NC1,C2);
END MODULE;

MODULE MAJORITY3_8;
INPUTS X.0,X.1,X.2,X.3,X.4,X.5,X.6,X.7,Y.0,Y.1,Y.2,Y.3,Y.4,Y.5,Y.6,Y.7,Z.0,
       Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,Z.7;
OUTPUTS MAJ-EXISTSP,A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7;
LEVEL FUNCTION;
DEFINE
G0(E0)
```

```
  = V-EQUAL_8(X.0,X.1,X.2,X.3,X.4,X.5,X.6,X.7,Y.0,Y.1,Y.2,Y.3,Y.4,Y.5,Y.6,
              Y.7);
G1(E1)
  = V-EQUAL_8(X.0,X.1,X.2,X.3,X.4,X.5,X.6,X.7,Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,
              Z.7);
G2(E2)
  = V-EQUAL_8(Y.0,Y.1,Y.2,Y.3,Y.4,Y.5,Y.6,Y.7,Z.0,Z.1,Z.2,Z.3,Z.4,Z.5,Z.6,
              Z.7);
G2A(ZERO.0,ZERO.1,ZERO.2,ZERO.3,ZERO.4,ZERO.5,ZERO.6,ZERO.7)
  = V-XOR_8(X.0,X.1,X.2,X.3,X.4,X.5,X.6,X.7,X.0,X.1,X.2,X.3,X.4,X.5,X.6,X.7);
G3(MAJ-EXISTSP) = OR3(E0,E1,E2);
G4(C.0,C.1,C.2,C.3,C.4,C.5,C.6,C.7)
  = TV-IF_8(E2,Y.0,Y.1,Y.2,Y.3,Y.4,Y.5,Y.6,Y.7,ZERO.0,ZERO.1,ZERO.2,ZERO.3,
            ZERO.4,ZERO.5,ZERO.6,ZERO.7);
G5(B.0,B.1,B.2,B.3,B.4,B.5,B.6,B.7)
  = TV-IF_8(E1,X.0,X.1,X.2,X.3,X.4,X.5,X.6,X.7,C.0,C.1,C.2,C.3,C.4,C.5,C.6,
            C.7);
G6(A.0,A.1,A.2,A.3,A.4,A.5,A.6,A.7)
  = TV-IF_8(E0,X.0,X.1,X.2,X.3,X.4,X.5,X.6,X.7,B.0,B.1,B.2,B.3,B.4,B.5,B.6,
            B.7);
END MODULE;
```

Appendix D. Mechanically Produced Schematics

Below we show the schematics produced by LSI Logic, Inc.'s design tool ‘‘liberate’’ from the NDL in Appendix C. We include the schematics for **`incrmt3`**, **`majority3`** and **`lstep_8`** only. These schematics are exhibited to emphasize the point that the NDL produced from our verified design can be processed by commercial design tools. This copyrighted material is used with the permission of LSI Logic, Inc.

DISCARD and replace with drawing

DISCARD and replace with drawing

DISCARD and replace with drawing

DISCARD and replace with drawing

DISCARD and replace with drawing

DISCARD and replace with drawing

DISCARD and replace with drawing

References

1. W.R. Bevier and W.D. Young. The Proof of Correctness of a Fault-Tolerant Circuit Design. Proceedings of the Second International Working Conference on Dependable Computing for Critical Applications, February, 1991, pp. 107-114.

2. R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, New York, 1988.

3. B.C. Brock and W.A. Hunt. A Formal Introduction to a Simple HDL. In *Formal Methods for VLSI Design*, J. Staunstrup, Ed., Elsevier Science Publishers B.V. (North-Holland), 1990, pp. 285-329.

4. J S. Moore. A Formal Model of Asynchronous Communication and Its Use in Mechanically Verifying a Biphase Mark Protocol. Tech. Rept. NASA CR-4433, NASA, 1992.

5. M. Pease and R. Shostak and L. Lamport. "Reaching Agreement in the Presence of Faults". *Journal of the ACM 27*, 2 (1980), 228-234.

6. J. Rushby and F. von Henke. Formal Verification of the Interactive Convergence Clock Synchronization Algorithm using EHDM. Tech. Rept. SRI CSL 89-3R, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, January, 1989.

7. D.E. Thomas and P. Moorby. *The Verilog™ Hardware Description Language.* Kluwer Academic Publishers, 1991.

Table of Contents